

Experiments in high precision clock synchronisation

Richard Jones

October 29, 1993

Contents

1	Introduction	1
2	Background	3
2.1	Existing methods	3
2.1.1	Deterministic Synchronisation	3
2.1.2	Network Time Protocol	4
2.1.3	Hardware Assisted Synchronisation	4
2.1.4	Synchronisation in a Transputer Network	4
2.1.5	Probabilistic Synchronisation	5
2.2	Recent proposals	5
2.3	Aims and objectives	6
2.3.1	New Features	6
2.3.2	Experimentation	7
2.3.3	New Algorithms	7
3	Behaviour of clocks of Departmental Suns	8
3.1	Message Passing Delays	8
3.2	Software Clock Problems	10
3.3	Clock Tick Losses	11
3.4	Drift Rates	12
4	Uoctimed	14
4.1	Description	14
4.2	Modifications	16
4.2.1	Message Passing Delays	16
4.2.2	Clock Tick Losses	18
4.3	Fault Tolerance	18
4.3.1	Slave failure	19
4.3.2	Clock failure	19
4.3.3	Master Server failure	19
4.3.4	Network Partition	20

5	Arvind's Algorithm	21
5.1	Description	21
5.2	Implementation	23
5.2.1	Methods Used to Calculate Parameters	23
5.2.2	Implementation of Master	24
5.2.3	Implementation of Slave	24
5.3	Experimentation	25
5.4	Extensions	29
5.4.1	Estimating master's time	29
5.4.2	Dynamic distributions	31
5.5	Comparison with Uoctimed	34
5.6	Modifications due to behaviour of clocks of Departmental Suns .	37
5.6.1	Message Passing Delays	37
5.6.2	Clock Tick Losses	37
6	Further Work	39
7	Conclusions	40

Chapter 1

Introduction

Hardware clocks used in computers tend to drift away from the correct time. In a distributed computer system, each hardware clock drifts away from the correct time at a different rate. In order to synchronise the clocks of each computer to the same time, some form of adjustment must be made to each clock. Internal clock synchronisation, where clocks can be adjusted so that all computers are synchronised to the same time relative to each other, is sufficient for some applications. Other applications require, external clock synchronisation, clocks are synchronised to some external time standard, such as Coordinated Universal Time (UTC), as well as to each other. Synchronisation between computers in the network is important for certain applications such as global ordering of events occurring throughout the system.

Clock synchronisation can be implemented using extra hardware devices or by software alone. Many algorithms proposed to synchronise the clocks in a distributed computer system. Most of the proposed algorithms are deterministic. Deterministic algorithms are based on message passing between nodes. They guarantee a maximum deviation between clocks of the nodes but this maximum deviation is limited by the maximum end-to-end message transmission delay which can be very large.

Recently some probabilistic algorithms proposed. These algorithms are still based on message passing but cannot **guarantee** a maximum deviation between computers in the network. Instead they guarantee a maximum deviation with a certain probability of failure. Such algorithms are not bound by the same constraint as the deterministic algorithms and can achieve much closer synchronisation.

In chapter 2 background information on clock synchronisation and the methods that have been published so far, at the end of the chapter the aims and objectives

of the project will be given. Chapter 3 describes discoveries that were made about the clock systems in the department. Chapter 4 gives information on the first of the two algorithms studied in this paper. Chapter 5 describes a new algorithm that has been implemented on the departmental suns. Chapter 6 describes further work that could carry on from this project. Chapter 7 gives the conclusions to the project.

Chapter 2

Background

2.1 Existing methods

Clock synchronisation in distributed systems has been an area of much research in recent years. Several published proposals are outlined in the following sections.

2.1.1 Deterministic Synchronisation

Many clock synchronisation algorithms are deterministic. They assume the existence of an upper bound on transmission delay. Since the upper bound is usually very large, deterministic algorithms cannot synchronise with certainty closer than

$$(d_{max} - d_{min})(1 - \frac{1}{N})$$

even in the presence of no failures and when clocks do not drift [5]. (d_{max} and d_{min} are the maximum and minimum transmission delays and N is the number of machines being synchronised).

A number of authors assume no upper bound on transmission delays and instead the timeouts for detecting communication failure are used as an upper bound. In [9] each node has an upper bound on its accuracy and nodes independently perform calculations across the network by collecting a set of times and making use of the response with the smallest error. This achieves an accuracy of $4(d_{max} - d_{min})$ which is approximately 50ms.

2.1.2 Network Time Protocol

Some experiments were done in [10] to determine the accuracy of Network Time Protocol (NTP) synchronisation on the internet. A client server model is used with the servers being connected by radio or wire to UTC. It was found that 30% of clocks were accurate to within 30ms of UTC, and over 90% were accurate to within 1 second. In [4] they attempted to obtain an upper bound on the accuracy of NTP over a Wide Area network, for example they used a complex datagram network between the U.S. and the U.K.

2.1.3 Hardware Assisted Synchronisation

There have been a number of hardware clock synchronisation solutions that can achieve a very close synchronisation but the extra cost is often prohibitive.

In [12], a compromise is presented where a software solution has minimal additional hardware to provide a balance between hardware requirements and the clock synchronisation attainable. The algorithm uses hardware to assist in accurate timing of message delays which gives the ability to achieve a synchronisation that is insensitive to the maximum variation in message delays. Software algorithms cannot do this.

The algorithm works on hypercube and mesh networks by each node determining the clock values on all other nodes and adjusting itself to the average of the others, while adjusting the clock on other nodes which have are very different from itself. The achieved synchronisation is in the order of 100-200 μ seconds.

2.1.4 Synchronisation in a Transputer Network

Transputer networks consist of many transputer components each having their own clock and connected to four neighbours by high speed communication lines. Proposed methods for synchronising the clocks in these networks ([15] and [7]), achieve a synchronisation of around 100 μ seconds. A complete network with a Hamiltonian circuit is required in [7] while [15] doesn't require a fully connected network.

The scheme proposed in [7] is a ring based synchronisation where the master periodically broadcasts to all the Processor Elements via the ring. This method is not fault tolerant but reduces the number of messages that must be exchanged around the ring.

In [15] processor elements (PE's) individually decide on their resynchronisation time and determine only the clocks of neighbours. A convergence function is

used to estimate the global time from direct neighbours. The actual rates of the clocks are adjusted in the algorithm so that rate differences between the clocks are reduced.

2.1.5 Probabilistic Synchronisation

Probabilistic algorithms are of most to us as they can achieve a high level of synchronisation. The design of *uocimed* came mainly from two probabilistic synchronisation methods, described in [8] and [5].

TEMPO is available for Sun 4 workstations which the University of Canterbury but it doesn't meet the accuracy requirements. TEMPO uses a master/slave scheme where the master sends time stamped messages to the slave and in reply the slave sends a time stamped reply to the master who then time stamps the message again. From these three time stamps the master estimates the time on the slave and transmits corrections. There is some fault tolerance built into TEMPO, for example if the master crashes or a network partition occurs then an election is held. TEMPO achieves an accuracy of 15ms between the master and slave and therefore 30ms between all machines on the network.

Cristian's algorithm was the first probabilistic algorithm and many other authors reference his paper and compare their performance to that of his algorithm. Similar to TEMPO Cristian's algorithm is also based on the master/slave principle. The master is closely synchronised to an external time source. Periodically the master attempts to synchronise with all the slaves. When a slave cannot achieve synchronisation within k attempts it leaves the group of synchronised slaves, but this only happens with a probability of $1-10^{-9}$. This achieves considerably better accuracy than TEMPO, with an accuracy of 1ms between master and slave (2ms between all machines on the network).

2.2 Recent proposals

Several recent clock synchronisation proposals that can be implemented at the University of Canterbury are outlined below.

A probabilistic clock synchronisation algorithm for large distributed systems is documented in [1]. It does not use the master/slave approach because of scalability problems. Instead a number of overlapping rings are used, with some nodes being in more than two rings to contain consistency throughout the system. Messages are sent around the rings with either an interval oriented or averaging approach used to estimate the correct time. The algorithm requires a large number of machines for it to be used effectively and since there are around 11 available machines in the department, it was not the most suitable choice.

A statistical synchronisation algorithm for Anisotropic Networks is presented in [6]. The aim is to calculate the relative drift rate and offset between two clocks. The protocol requires fast processors (so that the main application is not hindered by the synchronisation process) as it is computationally complex.

In [2] a new probabilistic algorithm is presented. The algorithm uses a master/slave protocol where the master sends multiple messages to the remote machine which averages the message times to estimate the time on the masters clock. It was claimed that the upper bound on the probability of invalidity decreases exponentially with the number of messages. This algorithm has similarities to *uotimed* but some new concepts were added which may improve on the existing implementation.

2.3 Aims and objectives

In previous work [3] *uotimed*, a high precision algorithm for clock synchronisation on Sun 3s had been developed. Also, some kernel modifications that enabled *uotimed* to run on the Sun 4s had been developed. The original aims of this project, discussed further below, were to port *uotimed* to the Sun 4 environment, to add a number of features to *uotimed*, to conduct experiments on the current version of *uotimed*.

2.3.1 New Features

Uotimed could be improved in several ways:

- The program could be made more fault tolerant.
- The clocks could be synchronised to an external time source such as Co-ordinated Universal Time (UTC).
- Currently a clock is adjusted by changing the value of its time. A system called tick adjustment can change the rate of the software clock and could be incorporated to slow the rate at which clocks drift relative to each other.

Also, *uotimed* could be made suitable for distribution to other sites by making it able to:

- run on machines other than Suns.

- handle Sun machines that have not had the kernel modified to increase resolution¹
- handle machines with limited clock resolution.

2.3.2 Experimentation

There is much experimentation that could be done on the current version of *uotimed*. One of the major areas that needs to be investigated is in the choice of values for the various parameters associated with *uotimed*. Many of these values have not yet been looked into in any depth. Other parts of the algorithm could also be investigated to determine the usefulness of certain methods, for example:

- What difference does kernel timestamping make?
- What difference does increased timestamp resolution make?
- How well would the algorithm work over Wide Area Network links?

2.3.3 New Algorithms

There has been other research work done in the area of clock synchronisation since *uotimed*. In particular some new algorithms for probabilistic clock synchronisation have recently been proposed. One of these, by Arvind, describes a new algorithm that could be implemented on the Computer Science system, thus enabling a comparison to be made between the new algorithm and *uotimed*. There are also other newly proposed algorithms with scope for implementation.

¹Timestamps t1 and t3 are recorded in the kernel rather than in the user process. Since timestamp t2 is already recorded in the kernel, all timestamps are recorded by the same software layer. This reduces the variable delay in message passing.

Chapter 3

Behaviour of clocks of Departmental Suns

If clocks are to be synchronised, it is necessary to have a good understanding of their behaviour. This chapter describes several aspects of the behaviour of the clocks of the Suns in the Department of Computer Science.

not word?

3.1 Message Passing Delays

The majority of clock synchronisation algorithms use message passing in order to estimate the time on a remote machine. Messages sent between two machines have a random time delay. There is a minimum delay, but usually there is no upper bound on the time taken. In Practice, most of the message delays are close to the minimum time with only a few delays being very large.

The two algorithms looked at in this report both use message passing. *Uoctimed* needs to keep track of the minimum message passing delay. In Arvind's algorithm, the mean delay is of great importance, it must be estimated accurately as it is used directly in the estimation of remote time.

Some interesting results were discovered by varying the time between sending the messages, and reading the average message delay. There appears to be a direct correlation between the interval between messages being sent and the message delays. The results of this experiment are shown in Figure 3.1. It can be seen that when the messages are sent with a very small interval between them, delays are smaller than when there is a large gap between messages.

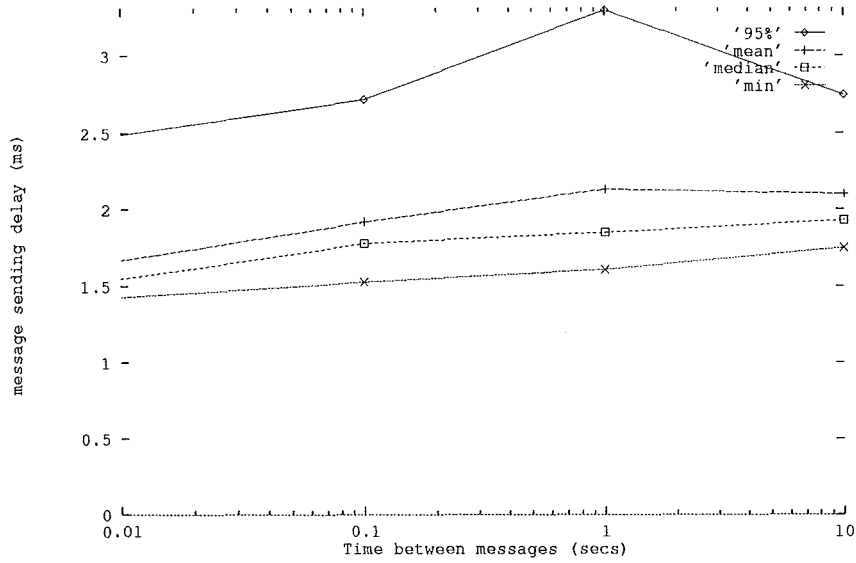


Figure 3.1: Message delays for the round trip time between kahu and hihi.

The machines kahu and hihi are on different segments of the network. In order to see if the phenomena described above occurs for machines on the same network segment, the above test was run between kea and kahu. These results are shown in Figure 3.2. Clearly the phenomena also occurs on machines on the same network segment.

The graphs do not show the maximum delay in the experiments as this is much larger than other values, the increases in the maximum delay were similar to that of the 95% values. The delays of all the parameters increased as the time interval increased to 0.1 seconds, and to 1 second. As the interval is increased further some of the parameters decreased, and stayed constant. Therefore there must be some time interval around 1 second where the relationship between message delays and time intervals no longer holds.

A possible explanation for this interesting relationship is that where intervals between messages are small, code and data related to message passing remaining in the CPU cache.

The effects on *uoftimed* and Arvind's algorithm are given in following chapters.

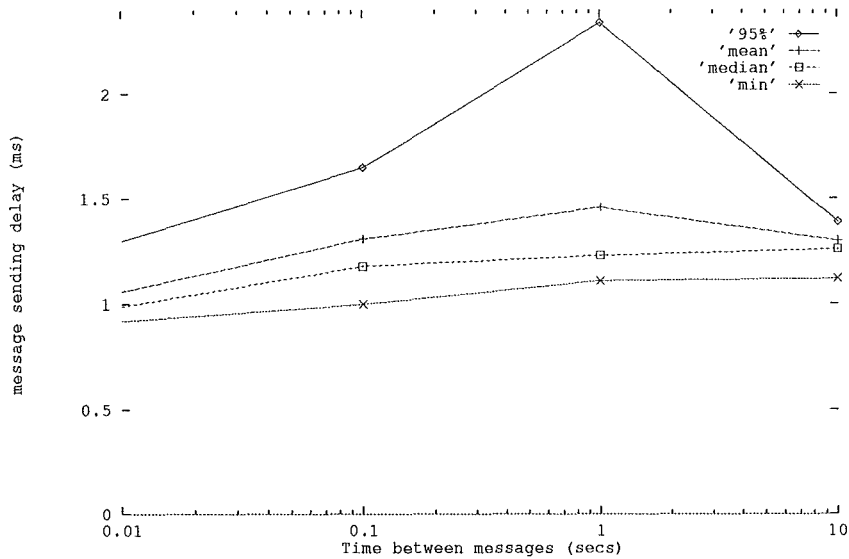


Figure 3.2: Message delays for the round trip time between kea and kahu.

3.2 Software Clock Problems

In order to keep clocks synchronised, clock values must be adjusted. This can be done by adjusting the hardware clock or by maintaining an offset to the hardware clock in software (creating a software clock). The second option was chosen as not all hardware clocks can easily be changed and it is quicker to change an offset in the software clock than to change the hardware clock.

To change the software clock the *adjtime* system call is used. This has the effect of speeding up or slowing down the clock so that time is a monotonically increasing function.

During initial testing of Arvind's algorithm it was found that after an adjustment to a clocks had been made, the clock would sometimes revert to its initial value after a few seconds though no other processes capable of changing the clock were running at the time.

It was eventually discovered that the kernel periodically tested to see if the software clock was out of synchronisation with the hardware clock by more than a predetermined tolerance. When this situation occurred the kernel would make an adjustment to the software clock to bring it back in line with the hardware clock. Disabling this solved the problem.

The reason for the this functionality in the kernel is to help improve the reliability of the software clocks. Problems with clock tick losses (described in the next section) were causing the software clocks to become unstable.

3.3 Clock Tick Losses

In the process of calculating the drift rates between the machines on the network certain properties about the clocks were discovered. Drift estimates are given in the next section.

To calculate drift rates a program called *checkdrift* was used. This program estimates the value of the clock on other machine by sending a message to them at the start and end of an interval. The drift rate is then determined by the following formula

$$\frac{(diff_end - diff_start)}{total_time}$$

where the *diff_end* and *diff_start* are the differences between the machines at the start and end of the period and the total time is the length of the period.

When estimating the drift rate between machines it is expected that they will be very similar in value over many periods, as it is assumed that two crystal controlled clocks will tend to drift from each other at an approximately linear rate. However this was found to not always be the case. There were numerous cases of 3-10 μ sec/sec differences from one hour to the next. Some cases of 80-90 μ sec/sec differences were noticed and in one extreme case the drift rate on kahu to three other machines changed from approximately -100 μ sec/sec one hour to 3170 μ sec/sec the next hour and back to the normal 35 μ sec/sec in the following hours. In the extreme case kahu drifted from other machines by around 11.5 seconds in one hour.

The program was modified by the author to ensure that the drift rates were being calculated correctly. The drift rate is now calculated by the formula.

$$\frac{local_time_end - local_time_start}{remote_time_end - remote_time_start}$$

With this modification to *checkdrift* the accuracy of drift estimates improved slightly but the main problem was still there.

Variations in drift appeared to happen predominately during work hours when the machines were the most heavily loaded. Another program *checkhost* was also used in conjunction with *checkdrift*. This program estimated the time on the remote clock by sending a message to the remote machine every second. With both programs running, drift variations could be located by *checkdrift* and investigated by *checkhost*.

Drifts	hihi	kahu	kaki	kea	kuku	moko	tete	titi	mohua	huia	kiwi
hihi	—	-21.7	0.6	-22.7	9.1	-9.4	12.5	2.4	99.3	114	121
kaku	21.7	—	22.3	-1.0	30.8	12.3	34.2	24.1	121	135	143
kaki	-0.6	-22.2	—	-23.3	8.5	-9.9	12.0	1.8	98.7	113	121
kea	22.7	1.0	23.3	—	31.8	13.4	35.2	25.1	122	136	144
kuku	-9.1	-30.8	-8.5	-31.8	—	-18.5	3.4	-6.7	90.2	104	112
moko	9.4	-12.3	9.9	-13.4	18.5	—	21.9	11.8	109	123	130
tete	-12.5	-34.2	-12.0	-35.2	-3.4	-21.9	—	-10.1	86.7	101	109
titi	-2.4	-24.1	-1.8	-25.1	6.7	-11.8	10.1	—	96.9	111	119
mohua	-99.3	-121	-98.7	-122	-90.2	-109	-86.8	-96.9	—	14.3	21.8
huia	-114	-135	-113	-136	-104	-123	-101	-111	-14.3	—	7.5
kiwi	-121	-143	-121	-144	-112	-130	-109	-119	-21.8	-7.5	—

Table 3.1: Drift rates between machines

It was determined that the drift variation was caused by machines loosing clock ticks when under heavy work load. The clock tick on a sun4 is 10ms and all the variations in time appeared to be 10ms or a multiple of 10ms. In one case, over a period of a couple of minutes the clock changed by around 11 seconds with some of the jumps as large as half a second. This accounted for the drift rates of $3170\mu\text{sec}/\text{sec}$ that were observed.

It was possible that the use of the console was causing the clock ticks to be lost, due to console commands that access the screen directly locking out other users. This was tested by running *checkhost* while performing display-related commands (such as *more* and *ls*). This did not cause any clock ticks to be lost but it did however cause the messages being sent to the machine to be greatly delayed or to timeout.

The loss of clock ticks can not be prevented. Any clock synchronisation algorithms will need to be able to cope with these problems. The effects on *uotimed* and *Arvind's algorithm* are given in the following chapters.

3.4 Drift Rates

Relative drift rates between the machines on the network is an important parameter in all clock synchronisation algorithms. The drift rate effects the length of the interval between resynchronisation attempts. To achieve the same level of synchronisation with a larger drift rate the interval between resynchronisations must be shorter.

As described in the previous section the drift rate on our network was calculated using *checkdrift*. Once the problems with clock tick losses were ironed out the relative drift values were calculated. The results are given in table 3.1.

It can be seen from the table that most of the machines have drift rates of up to $35\mu\text{sec}/\text{sec}$ relative to each other. Also the three sun 4m machines have drift rates $100\mu\text{sec}/\text{sec}$ greater than those of the other sun4 machines. If this was left unchanged synchronisation between the 2 machine types would be very difficult to perform.

The solution was to adjust the rate of the sun 4m machines clocks in line with the rest of the clocks. A method was found that adjusted the tick rate of the machines. It had a resolution of $100\mu\text{sec}/\text{sec}$ and once this was done the maximum relative drift rate between any two machines on the network was $44\mu\text{sec}/\text{sec}$, (between kiwi and kea).

Chapter 4

Uoctimed

4.1 Description

Uoctimed (University of Canterbury Time Daemon) is a clock synchronisation program developed to enable recording of interaction networks. *Uoctimed* uses a probabilistic algorithm based on Cristian's [5] and TEMPO [8], but with improvements outlined below. A greater level of accuracy was required for *uoctimed* than in other implementations with the requirement being a $100\mu\text{s}$ maximum allowable error.

The *uoctimed* process is run on all slave machines with the master machine simply returning the messages passed to it. Each slave independently decides when to attempt resynchronisation. The slave sends a message to the master which replies to the slave as shown in Figure 4.1. The message is timestamped in the kernel at the three points t_1 , t_2 and t_3 . The slave estimates the correction (c) required from these three timestamps with the formula:

$$c = t_2 - (t_1 + t_3)/2$$

In the perfect case, where the message passing delay is the same in both directions as shown in figure 4.1, the calculated correction would give an exact estimate of the master's time. If the message took longer in one direction as shown in Figure 4.2 then the slave would estimate the masters time to be half way between t_1 and t_3 giving the error in correction of (e) . To minimise the error only messages with a round trip time (rtt) less than a threshold value are accepted, this prevents messages with a large delay in one direction creating large errors in synchronisation.

A number of improvements have been made over the algorithms in Cristian's paper [5] and TEMPO [8], these are given in [3].

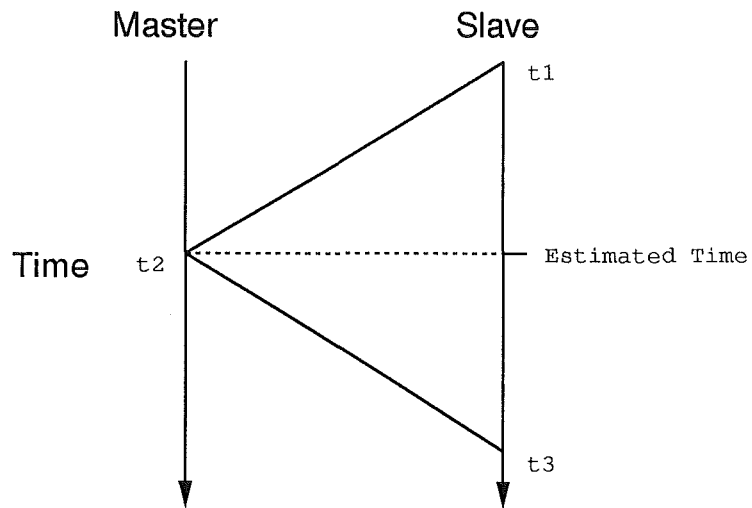


Figure 4.1: Message transmissions between master and slave in the perfect case.

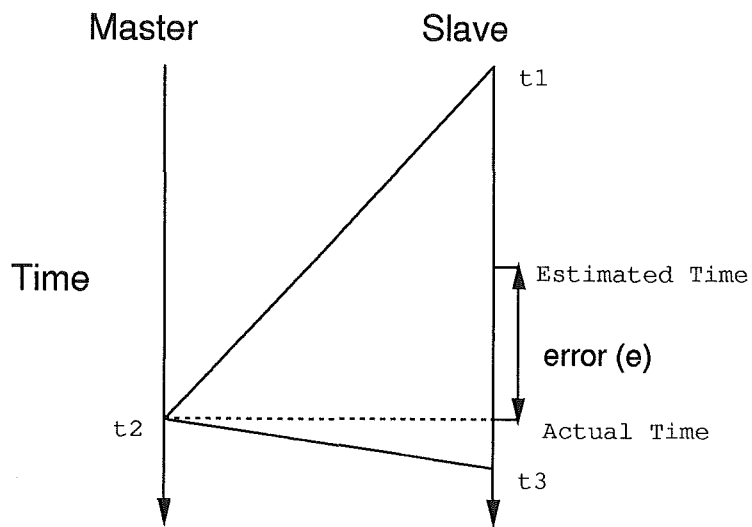


Figure 4.2: Message transmissions between master and slave in the worst case.

4.2 Modifications

In the previous chapter, discoveries were made about the clock systems at the University of Canterbury, some of which lead to modifications to *uotimed*, the implications of these are outlined below.

4.2.1 Message Passing Delays

During the experimentation in section 3.1 it was noticed that the message passing delays were dependent on the length of the interval between messages. The effect of this relationship on *uotimed* is described below.

In *uotimed*, the time interval between resynchronisation attempts is five seconds. At the end of each interval, a message is sent to the master, and a correction is calculated based on the timestamps in the reply. Messages are sent until the rtt is less than the threshold rtt additional messages are sent immediately after previous ones. Because there is no gap between messages during a resynchronisation attempt, the second and future messages will have considerably shorter transmission times than the first message since they will have had a message immediately before them, ie, with a very short interval.

The minimum transmission time is decreased whenever a message is sent with a shorter transmission time than the current minimum. The threshold rtt is dependent on the minimum return time. Whenever an attempt fails on the first message and a second message is sent, the second message will have a shorter rtt and may cause the minimum value to decrease. In future attempts at resynchronisation after an attempt with at least one failure, the minimum, and therefore threshold, rtt could be lower causing the first message of a resynchronisation attempt to be too long.

Experiments were conducted to test how these effects would influence the program. The author modified *uotimed* to output the minimum, threshold, and rtt for each message in a resynchronisation attempt. The program was run under normal load during work hours with a specified maximum error of 100μseconds. A section of the results is shown in Table 4.1.

Results show that in almost every case the rtt for the first message exchange was greater than the threshold rtt, and a second message (almost always less than the threshold value) was necessary. The size of the maximum error would influence these effects. The experiment was carried out with a small maximum error of 100μseconds. A larger maximum error would cause a higher threshold value which would cause the first message to be accepted more often, and would therefore reduce the number of messages required.

The overall effect on the performance of *uotimed* is that the number of messages

min	threshold	msg1	msg2	msg3
104	125	129	110	
104	125	128	110	
104	125	126	108	
104	125	128	108	
104	125	130	107	
104	125	131	109	
104	125	126	111	
104	125	133	110	
104	125	129	110	
104	125	136	108	
104	125	128	107	
104	125	132	108	
104	125	128	117	
104	125	128	108	
104	125	130	109	
104	125	133	115	
104	125	132	135	101
101	122	131	108	
101	122	119		
101	122	129	106	
101	122	134	108	
101	122	132	113	
101	122	131	110	

Table 4.1: Message rtt for uoctimed.

required can be up to twice as many as normal depending on the accuracy required.

4.2.2 Clock Tick Losses

It was noted in section 3.3 that under heavy load machines sometimes lose clock ticks. This has two major effects on *uotimed*; the clocks will suddenly lose synchronisation by a considerable amount and *uotimed* must resynchronise them, and the drift calculations made by *uotimed* may be affected by the missing ticks.

The original version of *uotimed* will resynchronise the clocks after a clock tick is lost, with the only problem being the time taken to do this. The lost clock tick(s) will be detected at the first synchronisation attempt using message passing. The *-m* option of *uotimed* can be used to make synchronisation attempts use message passing most of the time. If it is used to make most of the synchronisation attempts based on drift rate estimates, then it could take a long time before the problem is detected. There is no way of detecting earlier without more message passing and since lost clock ticks happen very rarely it is not considered a major problem. Once the lost clock tick(s) have been detected *uotimed* will start correcting the clocks again. How long this takes depends on the rate of adjustment used by the *adjtime(2)* system call. *Uotimed* was modified to use the *settimeofday(2)* system call if the adjustment required was larger than a specified amount. This was to resynchronise the clocks quickly if a number of clocks ticks were lost within a short period of time (see section 3.3), thus reducing the period that the clocks were out of synchronisation.

The drift calculations of *uotimed* involve each slave summing all adjustments made by it over an hour. If clock ticks are lost, the compensatory adjustments should not be included in the sum of adjustments. Each lost clock tick is easy to detect because it will cause a synchronisation loss of 10ms, whereas a normal synchronisation loss is less than one millisecond. *Uotimed* was modified to check if any adjustments were greater than a specified value and if so these adjustments were not included in the drift calculations.

4.3 Fault Tolerance

Uotimed currently has no provision for faults. There are four main types of faults that can occur, these are:

- Master failure — the machine being used as the master crashes.

- Network partition — loss of a communication link partitions the machines into two isolated groups, one of which contains the master.
- Slave failure — the machine running one of the slave processes crashes.
- Clock failure — the clock on one of the machines fails and gives false readings of the time.

Methods for coping with some of these types of faults have been considered in the paper by Cristian. The implications on *uocimed* of these faults have been are considered below:

4.3.1 Slave failure

If a slave fails then it will no longer be able to synchronise to the master. Since each slave is independent of the other slaves, when a slave crashes there will be no effect on any of the other slaves. The master is not affected, as it simply replies to messages from slaves. So the only effect will be that the slave will leave the group of synchronised machines.

4.3.2 Clock failure

Each slave should check for clock failure by considering the difference between master and slave clock times. If this is above a certain value then one of the clocks has failed, or the master or slave has lost some clock ticks. The slave can then check with other slaves to see if they have also experienced a clock failure. It could then be determined whether the master or slave clock had failed.

4.3.3 Master Server failure

Failure of the master could be determined by a slave when the master did not reply to several consecutive synchronisation messages. Once this is detected, the slaves would have to elect a new master.

There are different ways that this election process could be implemented. The current proposal is where a file is kept on the file system using a distributed method. This file contains a list of the slaves that could take over from the master in order, with the next slave to take over at the top. Once a slave detected the master had failed then it would start an election. One proposal for the election is where the slave sends a message to the slave on top of the list telling of the election, if this slave cannot be contacted then the next one on the

list is tried until a slave is found that could become the master. Once decided on then the new master tells all the slaves of the change in master.

A major problem with this method is that maintaining the file in a distributed method is not simple.

4.3.4 Network Partition

In the case of a network Partition a group of the slaves will still be able to contact the master and will continue to synchronise unaware that a network partition has occurred. The other group of slaves will no longer be able to contact the master and will have to hold an election between themselves. This will cause two groups of slaves synchronising to there master. When the network link returns then one of the two masters will have to change from a master back to a slave forming one group again.

Chapter 5

Arvind's Algorithm

This chapter presents an overview of an algorithm implemented at the University of Canterbury.

5.1 Description

Arvind's Algorithm [2], similar to that used by Cristian, involves a master/slave protocol where periodically the master sends messages to the slaves. The difference between Arvind's and Cristian's algorithms is the way in which corrections are calculated. In Arvind's algorithm, the master sends n timestamped messages to each of the slaves which timestamps each message on arrival. The slave then knows the time the message left the master and arrived at the slave, the slave takes the average of these message delays. The slave knows the expected delay \bar{d} for a message sent from the master to the slave. If the clocks were already synchronised the expected delay \bar{d} would equal the calculated delay from the messages, but if the clocks are not synchronised then the difference between the expected and calculated delay will be the difference between the clocks. The estimate of the master's clock at the instant the last message is received is then given by the formula:

$$T_{est} = R_n - \bar{R}(n) + \bar{T}(n) + \bar{d}$$

where R_n is the n th receive time, $\bar{R}(n)$ is the average of the n receive times, $\bar{T}(n)$ is the average of the n transmission times, and \bar{d} is the mean transmission time. A diagram of the message passing is shown in Figure 5.1. From the estimate of the master's clock and the time the last message was received the slave calculates the correction required and adjusts its own clock.

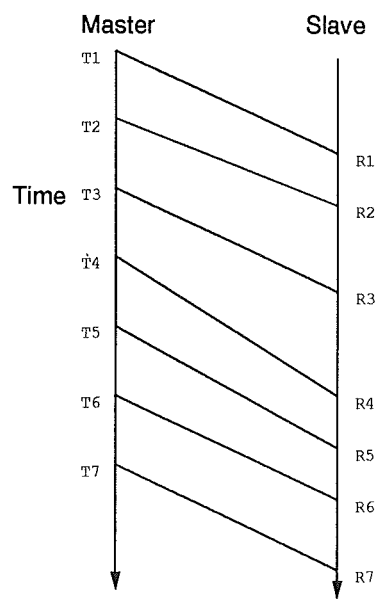


Figure 5.1: Message transmissions between master and slave in Arvind's algorithm.

n : the number of messages sent in one synchronisation attempt
 d_{min} : lower bound on the end-to-end message delay
 d_{max} : upper bound on the end-to-end message delay
 \bar{d} : average length of the end-to-end message delay
 d_{median} : median length of the end-to-end message delay
 σ_d : standard deviation of the end-to-end message delay
 ρ : relative clock drift rate
 ϵ_{max} : desired maximum transmission error
 γ_{ms} : maximum deviation between master and any slave
 p : probability of invalidity
 R_{synch} : interval between resynchronisation attempts
 τ : length of a resynchronisation attempt
 k : a parameter (dependent on p) that determines τ

5.2 Implementation

The author has implemented a version of Arvind's Algorithm that is written in C and runs on Sun 4 workstations. Important aspects of the implementation are discussed in this section.

5.2.1 Methods Used to Calculate Parameters

A number of terms and parameters used in the algorithm need to be specified or calculated. Values for these parameters were calculated as follows:

1. In order to calculate the parameters d_{min} , d_{max} , \bar{d} , σ_d , an experiment was performed where 5000 messages are sent between two machines with the rtt recorded (this experiment was used in [5] and [2]). The program *msgtrntimes* was written by the author to calculate the required values from the messages.
2. The relative drift rate between machines ρ is calculated using the existing program *checkdrift*.
3. The desired accuracy ϵ_{max} and γ_{ms} and probability of invalidity p need to be specified. From these the synchronisation period R_{synch} and the transmission period τ can be calculated.

5.2.2 Implementation of Master

The master machine runs a different program to that run on each slave machine. The basic algorithm for the master is given:

Master

initially

- locates the address of all the slaves.

continually

- Do
 - Do n times
 - A message is sent to each slave with no gap between messages
 - The master waits for a period of length $\frac{\tau}{n}$ (to spread the messages evenly over the entire interval τ).
 - Sleep for R_{synch}

5.2.3 Implementation of Slave

The basic algorithm for each slave is:

Slave

initially

- Determine average transmission delay.
- Synchronise reasonably close to master.

continually

- Do
 - Wait until a “Time is T_i ” message is received from master.
 - Do
 - Record the send and receive time from message.
 - Until n messages are received or the transmission period τ has ended
 - Calculate estimate of the master’s clock
 - Adjust local clock

Arvind’s algorithm required a different type of message than *uoc timed*. In *uoc timed* a protocol is used for sending messages and getting a reply so that the round trip time can be calculated. In Arvind’s algorithm messages are only sent in one direction with no reply, To allow this a new CLSYNC protocol was set up.

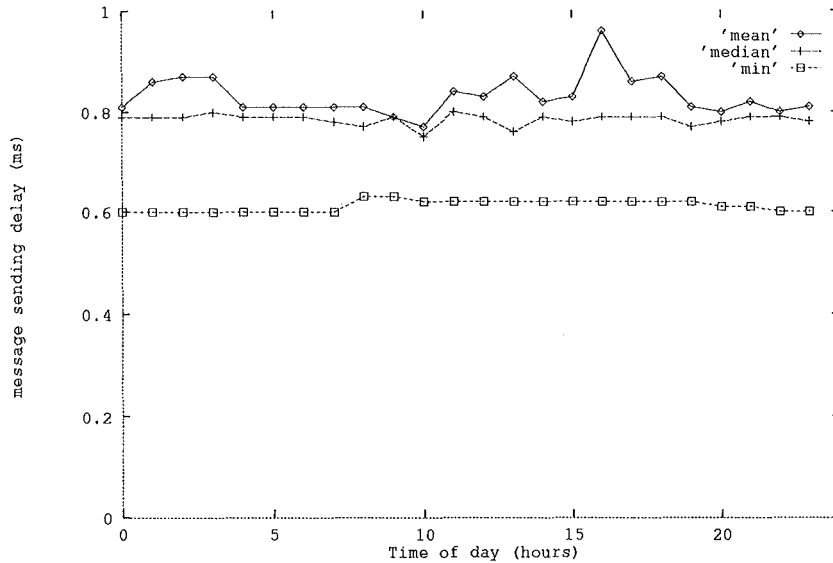


Figure 5.2: Message delays over a 24 period

5.3 Experimentation

A major assumption made in Arvind's algorithm is that the transmission time distribution is constant, implying the minimum, mean, and standard deviation of message delays are constant, usually due to the very light loads experienced on the machines. This was not thought to be the case for ethernet networks such as the departmental network of Suns, but in many cases the distribution between certain machines appeared constant. In order to test the effectiveness of Arvind's Algorithm under the conditions and assumptions of the paper, machines with a distribution that appeared to be constant were identified.

Four machines on the same network segment which generally have light loads on them were identified — kaki, hihi, moko, and tete. To determine the master, the relative drift rates of all four machines were calculated. Kaki was chosen to be the master as its drift rate was in the middle of the four machines.

To calculate the distribution of transmission times between the master and a slave, *msgtrntimes* was used to send 5000 messages from the master to the slaves and to record the message round trip times. The experiment was carried out 24 times with a one hour interval between experiments, over a period that included a working day. The results of the experiments are shown in Figure 5.2. Some variation occurs in the distribution, but it is reasonably close to constant. Distributions from the master to the other two slaves were very similar.

	Arvind's	Ours
ϵ_{max}	1ms	0.1ms
d_{max}	100ms	30ms
\bar{d}	2.45ms	0.445ms
σ_d	1ms	0.1ms
ρ	2 μ sec/sec	12 μ sec/sec
p	$1 - 10^{-6}$	$1 - 10^{-6}$
k	0.2	0.2
τ	100secs	2secs
n	24	24
γ_{ms}	4ms	0.2ms
R_{synch}	500secs	8.3secs

Table 5.1: Estimation of parameters

All of the parameters that are required by the algorithm are given in Table 5.1, with the values given in [2] included for comparison. The values were determined as follows:

- ϵ_{max} and γ_{ms} : In order to achieve a sub milli-second accuracy similar to that of *uotimed* 0.1ms and 0.2 ms were chosen.
- d_{max} , \bar{d} , and σ_d : were calculated from the experiments described above. Table 5.2 shows the results of the experiments where tete was the slave (the results are for the round trip time and need to be halved for the algorithm). The maximum from all three slaves was 27ms, this was taken to be d_{max} . The average values differed slightly between slaves, giving \bar{d} of 0.44ms, 0.415ms, and 0.445ms for hihi, moko, and tete respectively. The standard deviation σ_d is difficult to estimate. In Arvind's paper, they use the Gaussian distribution to approximate the end-to-end message delays (this approximation is justified in [2]). Using this approximation it can be shown that with a difference of 0.14ms between the median and the 95% quartile, the standard deviation σ_d is estimated at 0.07ms. The two other slaves had similar values and therefore σ_d is estimated at 0.1ms.
- ρ : from the drift calculations the maximum relative drift of the 3 slaves was 12 μ s per second.
- p : in *uotimed* the probability of invalidity is not mentioned since drift calculations are used if message passing fails. The same value as Arvind's was then chosen for p .
- k : a parameter used in calculating τ , defined as

$$k = \min(1, \frac{1}{\sqrt{2}(\text{erfc}^{-1}(p))^2})$$

	Arvind's	Tete
d_{max}	93.17ms	14.39ms
d_{min}	4.22ms	0.72ms
\bar{d}	4.91ms	0.89ms
d_{median}	4.48ms	0.83ms
95%	5.2ms	1.11ms

Table 5.2: Message passing delays between kaki and tete

where $erfc$ is a decreasing function of its argument. There is an error in Arvind's paper, as they give the above equation without the squared term around the $erfc$ function, and give a value for the result as if the squared sign is there. It seems more reasonable that the error is in the formula rather than the result so the above formula will be assumed in the rest of this report. The value of k is calculated as 0.2 for the given p .

- τ : the length of the period is given by the formula:

$$\tau << k \frac{\epsilon_{max}}{\rho}$$

the length of the period is considerably shorter than in Arvind's paper. This is due to the increased precision required and the drift rate between the departmental Suns being considerably larger than Arvind's drift rate.

- n : the minimum number of messages required is determined by formulae which depend on ϵ_{max} , σ_d , and p .
- R_{synch} : the synchronisation interval is calculated from the formula:

$$\gamma_{ms} = \epsilon_{max} + \rho(R_{synch} + d_{max} - d_{min})$$

The last two terms are usually insignificant. The length of the period is considerably shorter than Arvind's due, once again, to the higher drift rate and greater accuracy required.

With these values included in the algorithm an experiment was conducted to test its performance. Although the most accurate measure would involve using additional hardware to determine the time difference between the machines, the additional cost is prohibitive. Both *uocimed* and the implementation of Arvind's algorithm output the corrections that are made at each synchronisation attempt. In the ideal case of no errors all the corrections would be equal to the drift rate. Corrections that differ from the drift rate usually indicate periods of poor synchronisation.

At the instant before a correction is applied to a slave's clock, it has an error of $D \pm E$ where D is the drift error that accumulated over the synchronisation

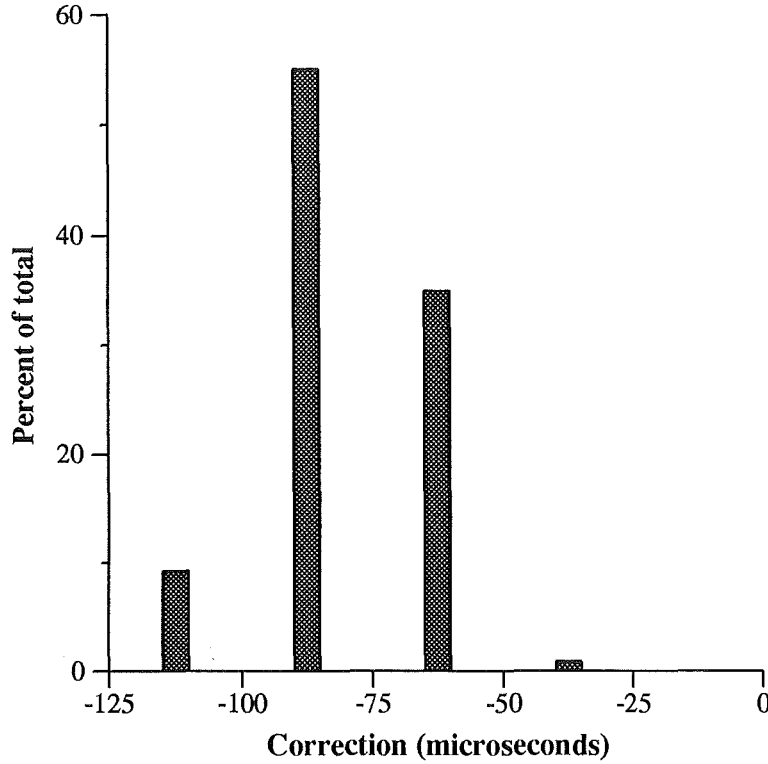


Figure 5.3: Distribution of correction values on moko when synchronised to kaki

interval, and E is the error in the synchronisation attempt. In resynchronising the slaves clock the range of corrections consistent with a maximum error of $E + D$ is, therefore, $D \pm 2E$.

The synchronisation program from Arvind's algorithm was run for approximately an hour. The results of the synchronisation from one of the slaves (moko) is given in Figure 5.3. The drift rate between kaki and moko is approximately $-10\mu\text{s/s}$, with a synchronisation interval R_{synch} the expected mean correction was $-80\mu\text{s}$. The observed mean correction was $-79\mu\text{s}$ confirming that the clocks stayed in synchronisation over the period. The range of corrections $-280\mu\text{s}$ to $120\mu\text{s}$ is consistent with the maximum clock synchronisation error of $\pm 180\mu\text{s}$. From Figure 5.3, 90% of the corrections were within $30\mu\text{s}$ of the mean, and 100% within $50\mu\text{s}$, all well within the maximum range, this shows that good synchronisation is achievable under light loads and a constant transmission distribution. Corrections made on the other two slaves had similar distributions, but were centred around different means (due to different drift rates).

5.4 Extensions

In the previous chapter Arvind's algorithm was implemented according to the specifications in his paper. In this section some extensions to his algorithm are made, the subsection 5.4.1 gives an improvement over the basic algorithm in arvinds paper. subsection 5.4.2 details how some of the restricting assumptions in Arvind's algorithm can be relaxed.

5.4.1 Estimating master's time

Investigation and experimentation into the distribution of message delays has shown trends about parameters in the distribution that may lead to modifications for Arvind's algorithm. In Arvind's algorithm the mean value of n messages from the master to the slave is used in calculating corrections. It can be seen from Figure 5.2 that the mean value of the distribution has greater fluctuations than the median and minimum values.

Investigations were made to see if Arvind's Algorithm could be modified to use the median or minimum value of the n messages from the master to the slave. Currently the master's clock is estimated by the formula:

$$T_{est} = R_n - \bar{R}(n) + \bar{T}(n) + \bar{d}$$

which is equivalent to:

$$T_{est} = R_n - avg + \bar{d}$$

where *avg* is the average message delay as measured from the n transmissions. This formula could be modified to:

$$T_{est} = R_n - median + d_{median}$$

where *median* is the median message length of the n transmissions. Another possibility is to use:

$$T_{est} = R_n - min + d_{min}$$

where *min* is the minimum message delay of the n transmissions.

The implementation was modified to incorporate the new methods of estimating the master's time, resulting in two additional programs and an experiment was conducted to determine their effectiveness. The experiment was similar to the experiment conducted in the previous section. The same four machines and the same parameters were used. The programs were consecutively run for half an hour.

The results of the experiments are given in Figure 5.4. The graph shows the corrections made on hihi, each column contains the number of corrections made

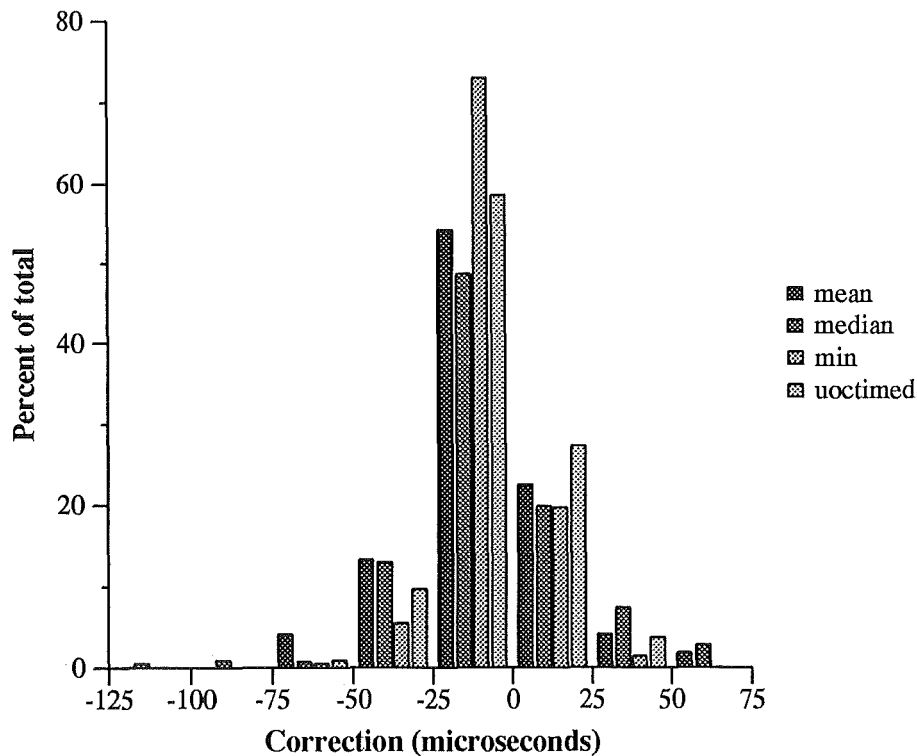


Figure 5.4: Distribution of correction values

within the interval by the given program. The mean value of the corrections was $-6\mu s$. In the ideal situation, all corrections would be at $6\mu s$, so the best method will have most corrections in the interval $-25\mu s$ to $0\mu s$ with less corrections in more distant intervals.

Comparing the three methods used here, the median method had the least corrections in the central interval and had corrections of over $-75\mu s$ and the most corrections above $25\mu s$. The minimum method had considerably more corrections in the central interval and only 2% of its corrections outside the central three intervals. The median method appears to be the worst with the mean next and the min method being a large improvement over the other two methods. The results on the other two slaves showed similar results.

The fact that the mean method was better than the median method was surprising. The median value is not influenced by very long messages as the mean value is. If some of the delays experienced are very large this should improve the median method relative to the mean method. The median values in Figure 5.2 appeared to be more stable than the medians, indicating that use of the median method would improve synchronisation. Figure 5.2 is based on sending 5000

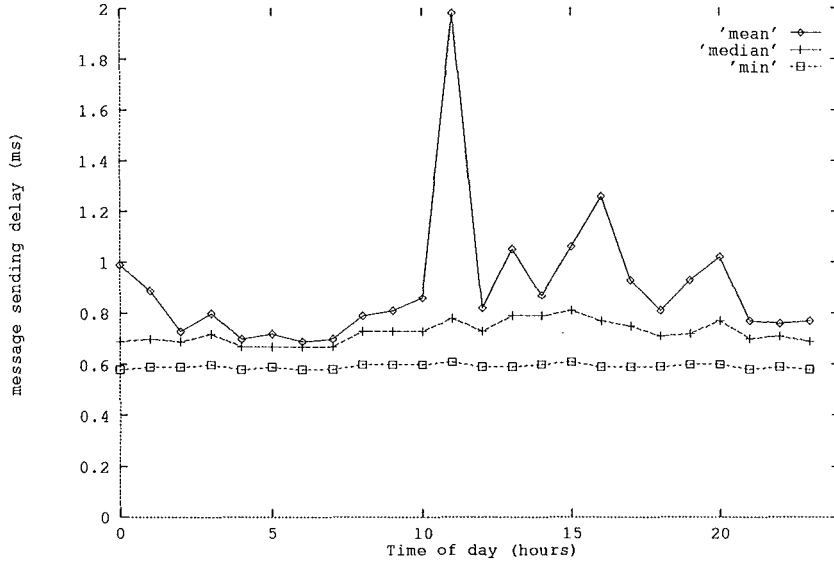


Figure 5.5: Message delays over a 24 period between kiwi and huia

messages in one sample, so the same effects may not occur for a smaller n such as the 24 used in this experiment.

From the results reported in this section, the minimum method is considerably better than the original mean method, so the minimum method is used in the version of Arvind's algorithm described in the rest of the report.

5.4.2 Dynamic distributions

A major assumption made in Arvind's algorithm is that the distribution of the end-to-end message delays is constant. This is not the case on an ethernet network such as the departmental sun network. In section 5.3 it was shown that certain machines had reasonably constant message delays. Other machines on the ethernet do not have constant distributions.

An experiment was run on the more heavily used machines (huia, kiwi, and mohua) to test how the distributions changed over time. The experiment was similar to that in section 5.3, where *msgsrntimes* sent 5000 messages 24 times with an hour between each group. The results from kiwi and huia are shown in Figure 5.5.

The experiment was run during the working week so that the main usage of the

machines is from 8am to 6pm. The changes in the transmission distributions reflect this, with very low mean values in the early hours of the morning and much higher values during working hours. It can be concluded from this that the message delay distribution is not constant over time.

The original version of Arvind's algorithm which used the mean of n messages from the master to the slave would perform very poorly under normal these conditions. The mean would have to be estimated at a value between 0.8ms and 1ms. This would cause large errors as, for example, in hour 11 in Figure 5.5 the error would be in the order of 1ms! The new version of Arvind's algorithm that uses the minimum delay in the distribution would perform considerably better. The minimum values appear to remain relatively constant with the maximum variation in the 24 hours being $30\mu s$. Although these results are for 5000 messages in smaller samples that were run the same trends appeared when smaller samples were run.

It has been shown that certain events can cause the minimum value to change, for example, when the screen became blank on the console the minimum value changed [3]. Therefore some form of determining the minimum value dynamically needs to be included.

Another assumption in Arvind's algorithm is that the transmission distribution on each slave is the same. This is not the case for an ethernet network such as the departmental network of suns, where the minimum round trip transmission delay has varied from 0.6ms to 1.2ms between different machines, with machines on different network segments having the largest values.

To incorporate the dynamic distributions the implementation of Arvind's algorithm was modified. The main modifications to the algorithm are:

- The estimation of the standard deviation σ_d and the drift rate ρ are done for all the slaves and defined in the program, from these other parameters such as the synchronisation periods and the number of messages n are calculated and defined in the program.
- Each slave independently calculates its value d_{min} . This is done initially by the slave sending a number of messages to the master, and from the round trip times of the messages the minimum value is estimated as half the round trip time.
- While the slave process is running, after each synchronisation attempt with the master the slave sends a round trip message to the master to test if the minimum transmission delay has changed.

The calculation of the standard deviation of message delays is not done dynamically, as it is difficult to estimate and a prohibitive number of messages

	kiwi	huia
d_{max}	195ms	13ms
d_{min}	1.27ms	0.72ms
\bar{d}	1.68ms	0.86ms
d_{median}	1.46ms	0.81ms
95%	2.7ms	1.04ms

Table 5.3: Message passing delays for kiwi to tete and huia to tete

would be required to keep a reasonably accurate estimate. It is likely to be more efficient to use more messages and achieve a better accuracy in general by assuming worst case standard deviations than to attempt to detect periods with lower standard deviations and to send fewer synchronisation messages in these periods.

Testing if the minimum transmission delay has changed in the slave is done in a similar way to *uoftimed*. If a reply is received in a shorter time than the current minimum then it is assumed that the minimum has changed and the new minimum is used. If a given number of messages are received in a row with a value greater than the minimum then the minimum is assumed to have increased and is adjusted. Determining the value for the number of messages in a row required is tricky. The value must be small enough so that increases are detected quickly and long enough so that there are few false alarms. If an incorrect value is used then synchronisation will be in error by the size of the error so it is important to find changes quickly.

Experiments were conducted to test the performance of the program under the new conditions. Five machines were chosen for the experiments — huia, kiwi, mohua, tete, and hihi. Tete was chosen to be the master as it's drift rate was in the centre of the relative drift rates of the machines. The most heavily loaded machines were included as were on both ethernet segments. The experiments were run during work hours so that normal loads were experienced.

To work out the parameters required *msgrntimes* was used. The results showed that machines on different ethernet segments had considerably longer delays, and the standard deviation of the delay distribution was also much larger. An example of the differences is shown in Table 5.3. Both kiwi and huia were connected to tete, kiwi is on a different segment of the ethernet and huia is on the same segment as tete. The results show kiwi to have much longer delays than huia. Estimating the standard deviations of both machines gave $\sigma_d = 0.06\text{ms}$ and $\sigma_d = 0.32\text{ms}$ for huia and kiwi respectively.

The relative drift rates were similar to the earlier experiments, so the resynchronisation intervals are the same as in section 5.3. The standard deviation is

different in these experiments, so the number of messages n was recalculated. The method for calculating n in Arvind's paper does not apply if minimum message delay is used. To determine message passing delays for kiwi to tete and huia to tete, four different experiments with different values of n (12, 24, 50, and 100) were run.

Results from the experiments show that changing the number of messages between the different values of n did not have a great effect on the size of the corrections that were made. What appeared to have a larger effect was the variations in the distribution over time. The same experiments were run on two different work days during work hours, and from the resulting corrections more variation in the synchronisation was observed from the changes in distribution than from changing the number of messages that were sent.

The results of the corrections on the slave kiwi in the worst of the two experiments on different days is shown in Figure 5.6. The mean correction was $-106\mu s$, 80% of the corrections were within 50ms of the mean and 100% of the corrections were within $135\mu s$ of the mean value. These values are consistent with the maximum error of 100ms.

The current minimum message delay and the length of the round trip delay of the message sent after each synchronisation attempt were also recorded. On the two slaves huia and hihi, both on the same ethernet segment as tete, the current minimum message delay only varied by $20\mu s$ and $30\mu s$. In both cases, many of the round trip message delays were equal to the minimum, so any error associated with incorrect estimates of the minimum message delay would be very small on those machines. On the other two slaves kiwi and mohua, the current minimum message delay varied by around $100\mu s$. The round trip message delays mainly varied from 0.65ms to 1ms, with few messages at the minimum value. It is unlikely that the actual minimum time varied by these amounts in the period of the experiment, so an additional synchronisation error would have been introduced. These errors could be reduced by either sending more round trip messages to the master (improving the accuracy of the estimate but increasing the overhead of the algorithm), or increasing the number of messages that must be received above the current minimum (would reduce the variation in the estimate, but would cause the detection time of an increase in the minimum value to be increased). Determining the actual errors in the estimation of the minimum can not easily be done without hardware assistance as the algorithm has no method of determining how accurate its estimate is.

5.5 Comparison with Uoctimed

The major motivation for the implementation of Arvind's algorithm was to allow comparison with *uoctimed*.

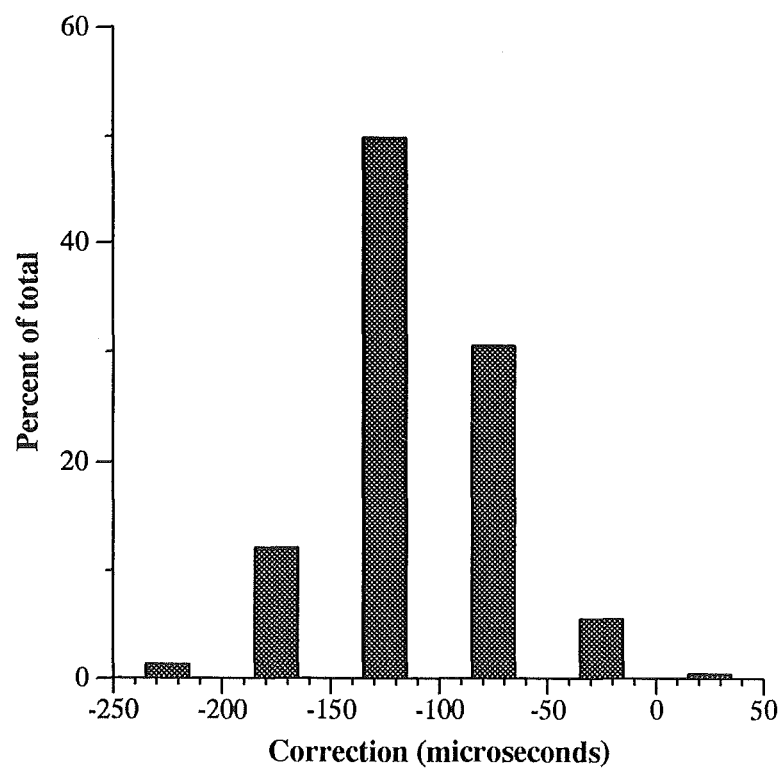


Figure 5.6: Distribution of correction values

In section 5.4, three different versions of Arvind's algorithm were tested, in addition to this *uotimed* was compared to the three different methods. In this experiment Arvind's algorithms were attempting to achieve an accuracy of $200\mu\text{s}$ between the master and slave. The required accuracy of *uotimed* is a command line option, therefore the accuracy was set to this value and run under the same conditions as Arvind's algorithm.

The results of the experiment are shown in Figure 5.2. *Uotimed* had more corrections in the central interval than the mean and median methods but much less than the minimum method. Less corrections were observed in the outer intervals than the mean and median methods, but more corrections than the minimum method. *Uotimed* performed better the mean and median but worse than the minimum method in relation to the accuracy that was achieved.

Uotimed did however have a significant advantage in the number of messages that were passed (which is the main cost of synchronisation algorithms). In *uotimed*, a resynchronisation attempt is made every 5 seconds and on average 4 messages are passed to estimate the correction required. In Arvind's algorithm, resynchronisation is attempted every 8 seconds and the actual synchronisation attempt takes 2 seconds, during which 24 messages are sent. Therefore Arvind's algorithm requires 2.4 messages per second, while in *uotimed* requires 0.8 messages per second, making *uotimed* much less costly in overhead.

In section 5.4.2 Arvind's algorithm was modified to work under transmission distributions that are not static, *uotimed* also works under these conditions. Comparisons should be made under these conditions of more normal workload. Making a fair comparison between the two algorithms was difficult to achieve because of the difficulty in estimating the achievable accuracy in Arvind's algorithm. *Uotimed* was therefore run twice, the first time aiming to achieve an accuracy of $200\mu\text{s}$ and the second time aiming to achieve an accuracy of $100\mu\text{s}$ between the master and the slave.

The results of the experiments showed that on the more heavily loaded machines such as kiwi, *uotimed* performed better than Arvind's Algorithm, but on the more lightly loaded machines such as hihi, Arvind's algorithm performed slightly better than *uotimed*. These conclusions are based on the correction distributions achieved.

In terms of the number of messages that each algorithm required *uotimed* came out best in every case.

5.6 Modifications due to behaviour of clocks of Departmental Suns

In Chapter 3 discoveries about the behaviour of the clocks of the departmental suns were made and in Chapter 4 the implications for *uotimed* were presented. This section describes the implications of these discoveries for Arvind's algorithm.

5.6.1 Message Passing Delays

During the experimentation described in section 3.1, it was noticed that the message passing delays were dependent on the length of the interval between messages. The effect of this relationship on *uotimed* is given below.

Arvind's algorithm determines the master's time by taking the average measured delay of the *n* messages sent by the master and subtracting the expected message delay. Messages are sent from the master with a constant interval between them. It is therefore important that when estimating the expected message delay that the same gap be used between the messages used to calculate the delay.

When the estimation was done using *msgtrntimes*, the gap between messages had to be set to the same length as the master would send. The most recent implementation that calculates the minimum message delay dynamically has been modified to calculate the correct gap between messages in the initial estimation. In the dynamic adjusting, two messages are sent to the master with only the second 'being used', (the first message supplies the correct gap before the second message).

When the master sends messages to the slaves, it cycles through each slave with no gaps between sending messages, a gap is then left before the next set of messages. The sending of many messages to different slaves with no gap between them may cause the differences in the message distributions to the slaves. This has not been tested and is left as future work.

5.6.2 Clock Tick Losses

In section 4.2.2 two major effects on *uotimed* were described. Drift rate calculations were affected and the algorithm had to be able to quickly resynchronise after losses. Arvind's algorithm does not calculate drift rates, so this was not a problem. In order to quickly resynchronise the clocks, the same method is used as in *uotimed*. A check is made at each correction and if the correction is larger

than a specified value then the *settimeofday(2)* system call is used rather than the *adjtime(2)* system call.

Chapter 6

Further Work

There were a large number of aims at the start of the project and there were also many ideas generated during the project; some of these have been addressed and some require further work. The sections that require further work are outlined:

- **Fault Tolerance** Idea's on how to implement fault tolerance for *uotimed* have been discussed, these ideas need formalisation and implementation.
- **Synchronisation with UTC** The possibility of synchronising the clocks to an external time source such as Coordinated Universal Time (UTC) could be looked into.
- **Improvements to *Uotimed*** There are still many area's of further investigation of *uotimed*. Some of these are:
 - Packaging *uotimed* so that it can be distributed to other sites.
 - Investigating and testing various parameters and methods used in *uotimed*
- **Improvements and testing of Arvind's algorithm** Extensive testing of Arvind's algorithm could be done under normal and high work loads. Investigating how more of the parameters could be automatically calculated, currently the minimum value is calculated dynamically but the other parameters have to be defined statically, making all the parameters dynamic would be an improvement. Being able to accurately determine the error bounds or the algorithm, currently determining the standard deviation is not easy and if the minimum value is used instead of the mean as suggested then a new method for determining the error bounds will have to be used.

Chapter 7

Conclusions

The original aims of the project were quite varied and due to time constraints only a few of them have been fully investigated.

A number of interesting discoveries were made about the clocks on the departmental Suns with some of these influencing the synchronisation algorithms either by changing their behaviour or making modifications necessary.

An implementation of Arvind's algorithm has been successfully completed. Some modifications were made that improved the performance of it, and some of the assumptions made in the paper were relaxed to make its use more general.

In comparison with *uotimed*, there were some situations under light loads that Arvind's algorithm achieved a closer synchronisation than *uotimed* but in most cases *uotimed* had the best performance, *uotimed* always required less message passing and therefore had less overhead than Arvind's algorithm.

Overall Arvind's algorithm can work effectively under the assumptions that he made, however some of these assumptions, such as a constant transmission distribution, make his algorithm very limited in use (in many real situations these assumptions do not hold). In this paper attempts have been made to relax Arvind's assumptions. Only limited success has been achieved due to the inherent difficulties in determining algorithm parameters.

Bibliography

- [1] ALAN OLSON, K. S. Probabilistic clock synchronization in large distributed systems. *11th International Conference on Distributed Computing* (1991), 290–297.
- [2] ARVIND, K. A new probabilistic algorithm for clock synchronization. *Proceedings. Real Time Systems Symposium* (1989), 330–339.
- [3] ASHTON, P., AND PENNY, J. Experiments with an Algorithm for High-Resolution Clock Synchronisation. In *Proceedings of the 15th Australian Computer Science Conference* (Hobart, Jan. 1992), Computer Science Association, pp. 41–55.
- [4] COLE, R., AND FOXCROFT, C. An Experiment in Clock Synchronization. *Computer Journal* 31, 6 (1988), 496–502.
- [5] CRISTIAN, F. Probabilistic clock synchronization. *Distributed Computing* 3, 3 (1989), 146–158.
- [6] D. COUVET, G. FLORIN, S. N. A Statistical Synchronization Algorithm for Anisotropic Networks. *Proceedings. Tenth Symposium on Reliable Distributed Systems* (1991), 42–51.
- [7] DE CARLINI, U., AND VILLANO, U. A simple Algorithm for Clock Synchronization in Transputer Networks. *Software—Practice and Experience* 18, 4 (April 1988), 331–347.
- [8] GUSELLA, R., AND ZATTI, S. The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3 BSD. *IEEE Transactions on Software Engineering* 15, 7 (July 1989), 847–853.
- [9] KEITH MARZULLO, S. O. Maintaining the Time in a Distributed System. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing* (1983).
- [10] MILLS, D. L. On the Accuracy and Stability of Clocks Synchronized by the Network Time Protocol in the Internet system. *Computer Communication Review* 20, 1 (April 1990), 65–75.

- [11] PARAMESWARAN, RAMANATHAN, KANG G. SHIN, R. W. B. Fault-Tolerant clock synchronization in distributed systems. *IEEE* (1990).
- [12] RAMANATHAN, P., KANDLUR, D. D., AND SHIN, K. G. Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems. *IEEE Transactions on Computers* 39, 4 (April 1990), 514–524.
- [13] SAMUEL J. LEFFLER, ROBERT S. FABRY, W. N. J., AND LAPSLEY, P. An Advanced 4.3BSD Interprocess Communication Tutorial.
- [14] SECHREST, S. An Introductory 4.3BSD Interprocess Communication Tutorial.
- [15] VERVOORT, W. A. TE WEST, R., SCHOUTE, A. L., AND HOFSTEDÉ, J. Distributed Time-Management in Transputer Networks. In *Proceedings of Euromicro '91 Workshop on Real-time Systems* (June 1991), IEEE, IEEE Computer Society Press, pp. 224–230.